1 Backpropagation

The purpose of this assignment is to implement a complete solution for training a neural network. In practice this is of course done by packages and programs, but it is important to have done it at least once yourselves.

Using backpropagation and stochastic gradient descent we will fit a nerual network with two hidden layers, and sigmoid activation, to the function $y(x) = x^3$ on the interval $(-2, 2)$. Let $\theta = \{(W_i, b_i), i = 1, 2, 3\}$ be the parameters of the model, then the network function is given by

$$f(x, \theta) = W_3 h_2 + b_3 \quad h_2 = \sigma(W_2 h_1 + b_2) \quad h_1 = \sigma(W_1 x + b_1)$$

1.1 Dimensions of the Parameters

Assume that both hidden layers has n hidden units, i.e. $h = (h1, ..., hn)$. Give the dimensions of all the parameters (W_i, bi), $i = 1, 2, 3$

$$\begin{aligned} W_1 &\in \mathbb{R}^{n\times 1} \quad b_1 \in \mathbb{R}^{n\times 1} \\ W_2 &\in \mathbb{R}^{n\times n} \quad b_2 \in \mathbb{R}^{n\times 1} \\ W_3 &\in \mathbb{R}^{1\times n} \quad b_3 \in \mathbb{R} \end{aligned}$$

1.2 Calculating the Derivatives.

Let $L(x_i, y_i, \theta) = (y_i - f(x_i; \theta))^2$. Given the observation (x, y) calculate the derivatives of $L(x, y; \theta)$.

$$\begin{aligned} \frac{\partial L}{\partial W_i^3} &= 2(y - f(x, \theta))h_i^2 \\ \frac{\partial L}{\partial b_i^3} &= 2(y - f(x, \theta)) \\ \frac{\partial L}{\partial h_i^2} &= \frac{\partial L}{\partial b_i^3} W_i^3 \end{aligned}$$

$$\begin{aligned} \frac{\partial L}{\partial W_{ij}^2} &= \frac{\partial L}{\partial h_i^2}(h_i^2(1 - h_i 2))h_j^1 \\ \frac{\partial L}{\partial b_i^2} &= \frac{\partial L}{\partial h_i^2}(h_i^2(1 - h_i 2)) \\ \frac{\partial L}{\partial h_i^1} &= \frac{\partial L}{\partial b_i^2} W_{ii}^2 \end{aligned}$$

$$\begin{aligned} \frac{\partial L}{\partial W_i^3} &= \frac{\partial L}{\partial h_i^1}(h_i^1(1 - h_i^1))x \\ \frac{\partial L}{\partial b_i^3} &= \frac{\partial L}{\partial h_i^1}(h_i^1(1 - h_i^1)) \end{aligned}$$

1.3 Fitting the Model

The following code fites a feed forward neural network with 2 hiddel layers (125 hidden units) to the function $y(x) = x^3$.

```
sigm <- function(x){
  f <- function(y){
    return(1/(1+exp(-y)))
  }
  return( sapply(x,f) )
}

initParams <- function(nrHiddenUnits, xDim) {
  W <- list()
  b <- list()

  # First hidden layer
  W_1 <- matrix(0.05*rnorm(nrHiddenUnits * xDim), ncol = xDim,
  nrow = nrHiddenUnits, byrow = TRUE)
  b_1 <- matrix(0.05*rnorm(nrHiddenUnits), ncol = 1,
  nrow = nrHiddenUnits, byrow = TRUE)

  # Second hidden layer
  W_2 <- matrix(0.05*rnorm(nrHiddenUnits * xDim), ncol = nrHiddenUnits,
  nrow = nrHiddenUnits, byrow = TRUE)
  b_2 <- matrix(0.05*rnorm(nrHiddenUnits), ncol = 1,
  nrow = nrHiddenUnits, byrow = TRUE)

  # Output layer
  W_3 <- matrix(0.05*rnorm(nrHiddenUnits), ncol = nrHiddenUnits,
  nrow = 1, byrow = TRUE)
  b_3 <- 0.05*rnorm(1)

  W[[1]] <- W_1
  W[[2]] <- W_2
  W[[3]] <- W_3

  b[[1]] <- b_1
  b[[2]] <- b_2
  b[[3]] <- b_3

  return(list("W" = W,"b" = b))
}


forwardPass <- function(x,W,b){
  #First hidden Layer
  h_1 <- sigm(W[[1]] %*% x + b[[1]])

  #Second hidden Layer
  h_2 <- sigm(W[[2]] %*% h_1 + b[[2]])

  #Output Layer
```

```
  y <- W[[3]] %*% h_2 + b[[3]]

  return( list("output" = y, "hiddenLayer1" = h_1, "hiddenLayer2" = h_2 ) )
}

gradient <- function(x,y,W,b){

  net <- forwardPass(x,W,b)
  h_1 <- net$hiddenLayer1
  h_2 <- net$hiddenLayer2

  output <- net$output[[1]]

  W_3 <- W[[3]]
  W_2 <- W[[2]]
  W_1 <- W[[1]]

  b_1 <- b[[1]]
  b_2 <- b[[2]]
  b_3 <- b[[3]]

  #dL/dW_3
  dW_3 <- 0*W_3
  for (j in 1:dim(dW_3)[2]) {
    dW_3[1,j] <- -2*(y-output)*h_2[j]
  }
  #dL/db_3
  db_3 <- -2*(y-output)
  #dL/dh_2
  dh_2 <- 0*h_2
  for ( i in 1:length(h_2) ) {
    dh_2[i] <- db_3*W_3[1,i]
  }


  #dL/dW_2
  dW_2 <- 0*W_2
  for (i in 1:(dim(dW_2)[1])){
    for (j in 1:(dim(dW_2)[2])) {
      dW_2[i,j] <- dh_2[i]*(h_2[i]*(1-h_2[i]))*h_1[j]
    }
  }
  #dL/db_2
  db_2 <- 0*b_2
  for (i in 1:length(db_2)) {
   db_2[i] <- dh_2[i]*(h_2[i]*(1-h_2[i]))
  }
  #dL/dh_1
  dh_1 <- 0*h_1
  for (i in 1:length(dh_1)) {
   dh_1[i] <- db_2[i]*W_2[i,i]
  }
```

```
  #dL/dW_1
  dW_1 <- 0*W_1
  for (i in 1:(dim(dW_1)[1])){
    for (j in 1:(dim(dW_1)[2])) {
      dW_1[i,j] <- dh_1[i]*(h_1[i]*(1-h_1[i]))*x
    }
  }
  #dL/db_1
  db_1 <- 0*b_1
  for (i in 1:length(db_1)) {
   db_1[i] <- dh_1[i]*(h_1[i]*(1-h_1[i]))
  }

  return( list("dW_1" = dW_1, "dW_2" = dW_2, "dW_3" = dW_3,
  "db_1" = db_1, "db_2" = db_2, "db_3" = db_3) )
}

backPropagation <- function(){
  learningRate <- 0.01
  params <- initParams(125,1)

  N= 100000
  for (iter in 1:N) {
    x <- -2+4*runif(1)
    y <- x^3
    grad <- gradient(x,y,params$W, params$b)

    params$W[[1]] <- params$W[[1]]-learningRate*grad$dW_1
    params$W[[2]] <- params$W[[2]]-learningRate*grad$dW_2
    params$W[[3]] <- params$W[[3]]-learningRate*grad$dW_3

    params$b[[1]] <- params$b[[1]]-learningRate*grad$db_1
    params$b[[2]] <- params$b[[2]]-learningRate*grad$db_2
    params$b[[3]] <- params$b[[3]]-learningRate*grad$db_3

    #if (iter %% 1000 == 0) print(iter)
  }
  return( params )
}

ans = backPropagation()
```

Now, we plot the predicted values of our model against the true curve $f(x) = x^3$. As we can see, the results resemble the true model really good.

```
call_forward_pass <- function(x){
  forwardPass(x,ans$W, ans$b)$output
}

df <- tibble(x = seq(-2,2,.05)) %>%
        mutate(y = map(x, ~call_forward_pass(.x)), y = as.numeric(y))

ggplot(df, aes(x,y)) +
```

```
geom_point() +
stat_function(fun=function(x) x^3, col = 'red') +
scale_colour_manual("Lgend title", values = c("red", "blue"))
```

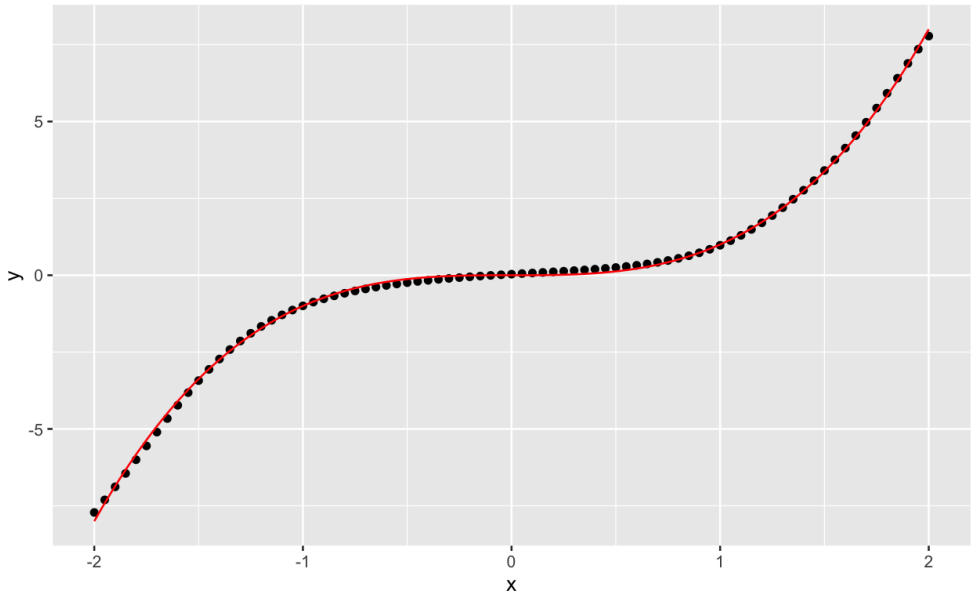

Figure 1:

2 Learning Keras

In this assignment we will fit a neural network for classifying images. We will use Keras, which is an extensive deep learning package in Python. Keras can be used in R through the keras package. But first we need to install it, see the steps below.

To get started with Keras follow the tutorial at `https://keras.rstudio.com/articles/getting_started.html`.

The documentationt to the fit function is availible at: `https://keras.rstudio.com/reference/fit.keras.engine`. training.Model.html. Look at the documentation and answer the following questions:

- Explain what a batch size is. - Explain what an epoch is. - Explain why the shuffle argument is default set to true, i.e. why do you want to shuffle? Can you think of example's when shuffling is a bad idea?

2.1 Batch Size

Is the number of samples that will be propagated through the network per gradient update, that is to say in one forward/backward pass. Some of the advantages of using a batch size smaller than the number of all samples are that it requires less memory and that the training process is usually faster because we update the weights after each propagation. A disadvantage

of using a batch size smaller than the number of samples is that the smaller the batch the less accurate the estimate of the gradient will be. In general we have

- **Batch Gradient Descent**. Batch Size = Size of Training Set
- **Stochastic Gradient Descent**. Batch Size = 1
- **Mini-Batch Gradient Descent**. 1 ¡ Batch Size ¡ Size of Training Set. Popular batch sizes include 32, 64, and 128 samples.

2.2 Epoch

In general, one epoch means that each sample in the training dataset has had an opportunity to update the internal model parameters, an epoch is comprised of one or more batches. A way to understand it is the following: we have a first for-loop over the number of epochs, in each loop we will a nested for-loop that iterates over each batch of sample.

In keras, the epoch parameter defines the number times that the learning algorithm will work through the entire training dataset.

2.3 Shuffle

This argument is a boolean that determines whether to shuffle the trining data before each epoch. The reason to set it by default is that we want the model to understand all inputs equally. For example, if the data is ordered by class, then the training result will be a bad one because the model will have a higher accuracy for the last class it trains than the first one.

Another reason why shuffle is true by default is that the objective function might have numerous minima, and therefore gradient descent algorithms are susceptible to becoming "stuck" in those minima. This is likely to occur if the set of observations is unchanged over all training iterations. Shuffling observations might help the solver to "bounce" out of a local minimum.

For the Batch Gradient Descent process (Batch Size = Size of Training Set) shuffling is unnecessary.

3 Fitting a Model in keras

Now we'll fit a feed-forward neural network to the dataset *dataset_fashion_mnist()*. For the following part python was used.

```
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D
from keras.datasets import fashion_mnist

from matplotlib import pyplot as plt

# Loading fashion_mnist dataset
fashion_mnist = keras.datasets.fashion_mnist
(train, test = fashion_mnist.load_data()
(train_images, train_labels) = train
(test_images, test_labels) = test
```

```
# Turning pixel values to values in the interval $[0,1]$
train_images = train_images / 255.0
test_images = test_images / 255.0

# Reshaping
train_images = train_images.reshape(-1,28,28,1)
train_images = train_images.astype('float32')
test_images = test_images.reshape(-1,28,28,1)
test_images = test_images.astype('float32')

# Neural net architecture
model = Sequential()

# Convolutional and max pooling layers since we are dealing with images
model.add(Conv2D(filters=64, kernel_size=2,
                 padding='same', activation='relu',
                 input_shape=(28,28,1)))
model.add(MaxPooling2D(pool_size=2))
# Dropout to deal with overfitting
model.add(Dropout(0.25))

# Repeat convolutional - max pooling - dropout
model.add(Conv2D(filters=32, kernel_size=2,
                 padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Dropout(0.25))

# Flattening
model.add(Flatten())

# Dense layer with 128 hidden units and relu activation
model.add(Dense(128, activation='relu'))

# Droput to deal with overfitting
model.add(Dropout(0.5))

# Output layer
model.add(Dense(10, activation='softmax'))

# Compiling model
model.compile(loss = 'sparse_categorical_crossentropy',
              optimizer = keras.optimizers.adam(),
              metrics = ['accuracy'])

# Training
model.fit(train_images, train_labels, epochs=10, validation_split=0.2)

# Evaluating
```

```
test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)
# Test error = 0.914

# Plotting validation curve
history = model.history
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='upper left')
plt.show()
```

The test error was .914 and the validation curve is given in the following figure:

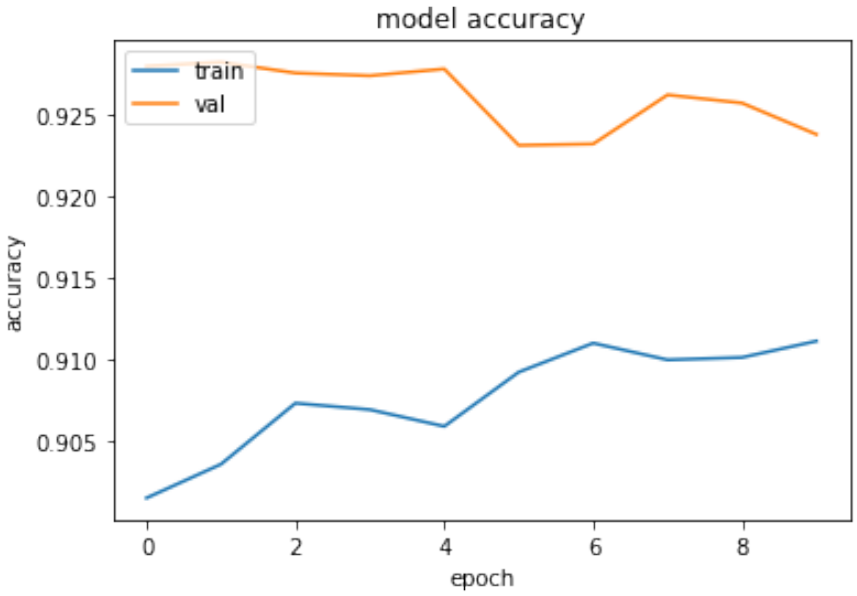

Figure 2: Validation curve